Towards Slice-Based Semantic Clone Detection

Hakam W. Alomari and Matthew Stephan
Department of Computer Science and Software Engineering
Miami University
Oxford, Ohio 45056

***Abstract*—This paper presents our proposed approach for detecting code clones based on similar slices of different versions of large software systems. We begin by presenting our initial thoughts on realizing software slice clone detection. We describe our initial results obtained by means of scripts to identify clones at different levels of granularity. The clones between versions are represented as pairs of cloned slices. Our results include a case study of over 191 versions of the Linux kernel, spanning over 10 years. In the near future, we plan on experimenting with established clone detectors to realize a complete and robust analysis approach.**

I. Introduction

Software maintenance of evolving large software systems is an ongoing challenge. Clone detection is one approach that can assist in maintaining systems by identifying recurring patterns of use. There exists many techniques for clone detection [1]. Most of them are either text- or syntactic-based. These techniques are good at detecting Type-1, Type-2, and Type-3 clones. However, they may miss some Type-4, semantic, clones. Type-4 clone identification is challenging as it can be difficult to determine precisely semantically similar components [2], [3]. To help understand semantic similarities, program slicing is a widely used and well-known approach for comprehending and detecting semantic properties of software [4], [5].

In this position paper, we consider semantic clone detection by means of slice similarity. We consider two code fragments semantically similar if their slices are similar. We present our ideas in the form of our early experiments in detecting clones in program slices on the Linux kernel across different versions. Using our scalable slicer, srcSlice [5], we identify clones through our existing slice-based metric called ”hashSize” [6]. We detect clones at three different granularity levels. We present our initial results and future plans in order to garner suggestions and collaborate within the cloning community.

II. Related Work

Using text-based approaches to detect structural similarities between program versions has obvious utility in software evolution. However, results of existing tools may be unsatisfactory because their view is based on program text or tokens only rather than program behavior [1]. They may be sensitive and unable to identify clones that are substantially modified structurally, but equivalent semantically.

There exists syntactic-based source code clone detection techniques that are tree- and graph- based. Tree-based techniques compare Abstract Syntax Trees (ASTs) of code fragments to detect clones. Graph-based techniques compare Program Dependence Graphs (PDGs) using sub-graph isomorphism to detect source code clones. Both types of these techniques do not scale well for multiple versions of large software systems [5] nor are able to find Type-4 clones.

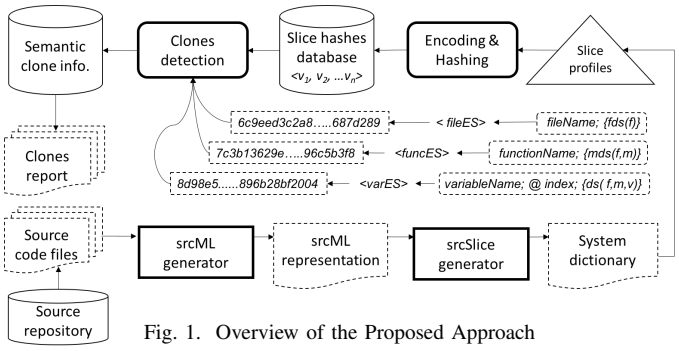

Fig. 1. Overview of the Proposed Approach

Other metrics-based techniques calculate different metrics from the source code and compare these metric values to indicate cloned fragments [1]. These techniques compute metrics using only syntactic information of the source code and use that to model semantic information. For example, they compute cyclomatic complexity by counting the number of conditionals/branches to infer semantic complexity. Semantic information can be much more difficult to derive and model.

III. Proposed Approach

As we show in Figure 1, we begin by downloading all consecutive versions of the Linux kernel from its subversion repository. We then convert the source code into srcML format [7] in order to use srcSlice to build a system dictionary with slice profiles for all slicing variables. srcSlice is a slicing tool implementing forward, static, inter-procedural, and decompositional slicing [5]. srcSlice computes a slice profile line by line for each variable as it is encountered. After srcSlice computes all the slice profiles, it performs a single pass through all profiles to take into account dependent variables, function calls, control-flow edges, and direct pointer aliasing to generate the final slices.

We encode the slice profiles to strings and feed those encoded strings to an MD5 [1] hash algorithm to produce a 128-bit hash value, which we express as a 32 digit hexadecimal number. The encoding process considers different granularity of slices: variables, functions, and files. We compare slice hashes for all granularity across all versions. varES, funcES, and fileES represent the number of variables, functions, and files that have similar slice profiles across versions, respectively. Ideally, the comparison process will have linear time

[1] Available online: http://www.ietf.org/rfc/rfc1321.txt

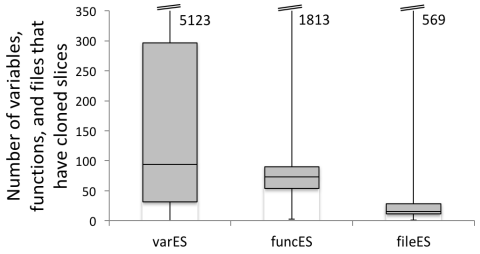

Fig. 2. Boxplot for varES, funcES, and fileES of 191 Linux kernel versions. The upper line values exceed the top of the graph and their true values appear on the labels above their respective up-bar boxes.

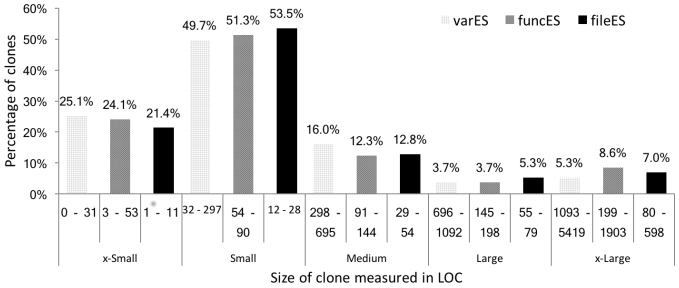

Fig. 3. Histogram for distribution of clones for 191 Linux kernel versions.

complexity since the hashed values are hexadecimal numbers. We recommend saving clone pair information in a database for further analysis.

IV. Discussion

In this section we present our initial results along with our motivation of why, how, and whom slice-based clone information can help. Our hypothesis and hope is that this information can serve as a complement or supplement to existing maintenance and comprehension approaches. We believe that in order to help understand all types of clones, we need to raise the level of abstraction beyond simply syntactic clones. This can be achieved by the decomposition/composition mechanisms employed in our srcSlice approach. In our case, a sequence of lower-level clones, that is, similar slices at the variable-level, are composed to form a single higher level clone at the function and file levels to encapsulate semantics. For example, implementing a new feature is comprised of all small clones that a programmer performed to develop the feature. These clones include any refactoring and modifications which are needed to achieve the goal.

A. Motivation and Research Questions

We have begun considering a number of research questions: (RQ1) what are some of the cloning patterns, (RQ2) what are the typical size of clones, and (RQ3) how common are the clones. Future questions we are considering include (RQ4) which clones can be detected only using slice-based clone detection, (RQ5) who could benefit from slice-based clone detection, and (RQ6) what will be the precision and recall of slice-based clone detection approaches.

B. Initial Results

We perform clone pair detection using comparison scripts. We apply three measures of semantic clone detection to 191 releases of Linux kernel, spanning 10 years. We use the values of these measures as data points for classification. We first take a holistic view of all clones in subset of the history, and then use descriptive statistic methods to classify them. We employ five quartiles for categorization: Q0 to Q4. We use Inter-Quartile Range (IQR = Q3 - Q1) as a measure of spread. It is helpful in identifying outliers. We used the 1.5 x IQR rule: Anything below Q1 - (1.5 x IQR) or above Q3 + (1.5 x IQR) are outliers. The Boxplot graph in Figure 2 shows the median (Q2) as the line through the middle, quartiles, and outliers. Each box illustrates the region between Q1 and Q3. The lines leaving the boxes show the range of values that are not outliers.

Based on our classification, we distribute our data into five regions representing classes of clone size for each granularity: extra-small, small, medium, large, and extra-large. We present this in Figure 3 for each of the varES, funcES, and fileES measures. For example, the percentage of clones at the variable level that have a size $<= 31$ LOC is equal to 25.1%. We first observe most of the clones are small or extra small. Specifically, for all three granularities, approximately 75% of the clones are small or extra-small. However, notable larger clones also exist. Larger clones are often those that touch almost every artifact in the system. Our manual inspection of selected versions indicates most commonly occurring clones were those related to initialization of new hardware drivers. Adding new drivers was often associated with the introduction of new clones. One possible explanation is that developers used existing tested and working driver code rather create it from scratch.

V. Conclusion

We presented our ideas and initial results for detecting semantic clones using program slices. We believe this is a new perspective on semantic clone detection, allowing for large system and multiple-versions cloning analysis. Future work involves experimenting with various established clone detectors, such as Nicad and CCFinder. We plan to continue this line of research and provide semantic clone information that can serve as a complement to current syntactical-based approaches to help maintainers understand program clones across versions.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queens School of Computing, Tech. Rep. 2007-541, 2007.

[2] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*. Washington, DC, USA: IEEE, 2007, pp. 96–105.

[3] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*. New York, NY, USA: ACM, 2008, pp. 321–330.

[4] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[5] H. W. Alomari, M. L. Collard, J. I. Maletic, N. Alhindawi, and O. Meqdadi, "srcslice: Very efficient and scalable forward static slicing," *J. Softw. Evol. Process*, vol. 26, no. 11, pp. 931–961, Nov. 2014.

[6] H. W. Alomari, M. L. Collard, and J. I. Maletic, "A slice-based estimation approach for maintenance effort," in *ICSME*. IEEE, 2014, pp. 81–90.

[7] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *ICSM*. IEEE, 2013, pp. 516–519.